Zawody III stopnia

opracowania zadaĆń

Piotr Chrząstowski-Wachtel **Marcin Kubica**
treść zadania, opracowanie program wzorcowy

Alibaba

Aby otworzyć Sezam, trzeba mieć komplet co najmniej z żetonów złotych, s — srebrnych i m — miedzianych. Alibaba ma początkowo pewną liczbą żetonów każdego rodzaju i może je wymieniać u strażnika Sezamu według ściśle określonych reguł.
Każda reguła jest postaci:

$$z_1, s_1, m_1 \to z_2, s_2, m_2 \qquad (z_i, s_i, m_i \in \{0, 1, 2, 3, 4\})$$

gdzie: z_1, s_1, m_1 oznaczają odpowiednio liczby żetonów złotych, srebrnych i miedzianych, jakie Alibaba musi dać strażnikowi, zaś z_2, s_2, m_2 — liczby żetonów złotych, srebrnych i miedzianych, które otrzyma w wyniku transakcji wymiany.
Żetony otrzymane w wyniku transakcji, można wymieniać w kolejnych transakcjach.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego ALI.IN zestawy danych, z których każdy zawiera:*
  - *liczby żetonów złotych, srebrnych i miedzianych znajdujących się początkowo w posiadaniu Alibaby,*
  - *opis kompletu żetonów otwierającego Sezam, oraz reguły transakcji;*
- *dla każdego zestawu danych stwierdza, czy istnieje skończony ciąg transakcji, który pozwoli Alibabie zgromadzić komplet żetonów otwierający Sezam i jeśli tak, znajduje i zapisuje w pliku tekstowym ALI.OUT minimalną długość takiego ciągu, a w przeciwnym przypadku zapisuje w pliku ALI.OUT odpowiedź NIE.*

Wejście

W pierwszym wierszu pliku tekstowego ALI.IN jest zapisana jedna liczba całkowita dodatnia $d \leqslant 10$. Jest to liczba zestawów danych.
Dalej są zapisane kolejno zestawy danych. Każdy zestaw danych składa się z wielu wierszy.
W pierwszym wierszu są zapisane trzy liczby całkowite nieujemne $z_p, s_p, m_p \in \{0, 1, 2, 3, 4\}$. Są to liczby żetonów złotych, srebrnych i miedzianych, jakie na początku ma Alibaba.
W drugim wierszu są kolejne trzy liczby całkowite $z, s, m \in \{0, 1, 2, 3, 4\}$. Są to liczby żetonów złotych, srebrnych i miedzianych, jakie trzeba mieć, aby otworzyć Sezam.
W trzecim wierszu jest zapisana liczba reguł r, gdzie $1 \leqslant r \leqslant 10$.
W każdym z kolejnych r wierszy jest zapisany ciąg sześciu liczb $z_1, s_1, m_1, z_2, s_2, m_2$, gdzie $z_i, s_i, m_i \in \{0, 1, 2, 3, 4\}$. Każda taka szóstka liczb jest zapisem jednej reguły transakcji: $z_1, s_1, m_1 \to z_2, s_2, m_2$.
Liczby w każdym wierszu są pooddzielane pojedynczymi odstępami.

Wyjście

W i-tym wierszu pliku tekstowego ALI.OUT należy zapisać wynik dla i-tego zestawu danych:

- *jedną liczbę całkowitą nieujemną, tj. minimalną liczbę transakcji wymiany, jakich Alibaba musi dokonać, aby zgromadzić komplet żetonów otwierający Sezam,*
- *albo jedno słowo NIE.*

Przykład

Dla pliku tekstowego ALI.IN:

```
2
2 2 2
```

```
3 3 3
3
0 1 1 2 0 0
1 0 1 0 2 0
1 1 0 0 0 2
1 1 1
2 2 2
4
1 0 0 0 1 0
0 1 0 0 0 1
0 0 1 1 0 0
2 0 0 0 2 2
```

poprawnym rozwiązaniem jest plik tekstowy ALI.OUT:

```
NIE
9
```

Twój program powinien szukać pliku ALI.IN w katalogu bieżącym i tworzyć plik ALI.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę ALI.??? gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonalnej powinien być zapisany w pliku ALI.EXE.

ROZWIĄZANIE

Zadanie o Alibabie było jednym z łatwiejszych zadań trzeciego etapu. Kilkunastu uczestników uzyskało za jego rozwiązanie maksymalną liczbę punktów, co nota bene nie zawsze oznaczało bezbłędność rozwiązania. Zadanie przypominało te zadania z poprzednich olimipiad, które rozwiązywało się za pomocą przeszukiwania grafu wszerz. Wytrawni uczestnicy, gdy widzą w treści sformułowanie, które wymaga znalezienia najkrótszej drogi prowadzącej do jakiegoś celu, od razu zastanawiają się nad grafem, który można by zbudować, aby zastosować znany algorytm.

Większość uczestników wpadła na to, że tym razem węzłami grafu będą trójki liczb określające liczbę poszczególnych rodzajów żetonów, które akurat są w posiadaniu Alibaby.

Tym razem główna trudność polegała na tym, żeby zatrzymać przeszukiwanie grafu w odpowiednim momencie. Graf bowiem potencjalnie był nieskończony i, w przypadku odpowiedzi negatywnej, bez trudu można było wpaść w nieskończoną pętlę. Mogło się bowiem zdarzyć, że Alibaba przy zadanych regułach nie miał szans dostać się do Sezamu, choć mógł wykonywać nieskończoną sekwencję ruchów tworząc coraz to nowe konfiguracje.

Przykładem takiej sytuacji były następujące dane: Alibaba ma początkowo jeden żeton złoty. Jeden żeton złoty można wymienić na jeden srebrny lub jeden brązowy. Jeden srebrny można wymienić na dwa srebrne, a jeden brązowy na dwa brązowe. Aby otworzyć Sezam trzeba mieć jeden żeton srebrny i jeden brązowy. Widać, że Alibaba może uzyskać dowolną liczbę żetonów srebrnych bądź brązowych, ale nie może mieć jednocześnie i srebrnego i brązowego. Graf powstający przez stosowanie powyższych reguł jest nieskończony i naiwne przeszukiwanie wszerz prowadziłoby do zapętlenia.

Co można zatem zrobić, żeby uniknąć nieskończonej pętli w algorytmie? Istnieją co najmniej dwie metody. Pierwsza, to uciąć przeszukiwanie, gdy tylko zorientujemy się, że dalsze szukanie jest już beznadziejne. Tę metodę stosowała większość zawodników, z różnym zresztą skutkiem. Podstawowa trudność polega na rozpoznaniu sytuacji, w której już dalej szukać nie warto, bo wiadomo, że rozwiązania dalej na badanej ścieżce być nie może.

Najbardziej naturalnym pomysłem jest w tej metodzie ograniczenie wartości współrzędnych. Chodzi o wyznaczenie takiej wartości, żeby nie było konieczne dalsze rozważanie ścieżki, jeśli taką wartość osiągniemy na którejś współrzędnej. Taką wartością było przy przyjętych ograniczeniach danych 100. Jeżeli któraś ze współrzędnych osiągnęła 100, to dalsze szukanie było bezcelowe. Oto dane, dla których możliwe było osiągnięcie 100:

- początkowa konfiguracja Alibaby: $(0, 0, 4)$,

- reguły wymiany: $(0,0,0) \to (0,1,4)$, $(0,4,0) \to (1,0,4)$
- konfiguracja otwierająca Sezam: $(4,4,0)$.

Łatwo zauważyć, że aby osiągnąć wartość 4 na pierwszej współrzędnej, trzeba czterokrotnie skorzystać z drugiej reguły. Zastosowanie jej wymaga każdorazowo zgromadzenia 4 żetonów srebrnych, a wygenerowanie każdego z nich wymaga czterokrotnego zastosowania reguły pierwszej. Podsumowując, aby uzyskać jeden złoty żeton, trzeba czterokrotnie skorzystać z reguły pierwszej i raz z drugiej. Każde zastosowanie którejkolwiek z reguł zwiększa liczbę żetonów brązowych o 4. Wygenerowanie jednego żetonu złotego wiąże się zatem, czy chcemy tego, czy nie z otrzymaniem 20 żetonów brązowych. Po uzyskaniu w ten sposób 4 żetonów złotych mamy więc 84 żetony brązowe i 0 srebrnych. Pozostaje jeszcze czterokrotnie zastosować regułę pierwszą, aby uzyskać dodatkowo 4 żetony srebrne (i przy okazji 16 brązowych). Kończymy więc nasz algorytm mając 4 żetony złote, 4 srebrne i 100 brązowych.

Mając to ograniczenie 100, modyfikujemy algorytm przechodzenia grafu wszerz w następujący sposób. Zaczynamy od konfiguracji początkowej, którą, o ile nie otwiera od razu Sezamu, wkładamy do kolejki etykietując ją liczbą 0 (etykieta będzie oznaczała minimalną liczbę zamian doprowadzającą do związanej z nią konfiguracji). Następnie, aż do momentu, w którym znajdziemy element otwierający Sezam, lub kolejka nie stanie się pusta, wykonujemy następujące operacje:

(1) Pobierz z kolejki kolejny element v wraz z towarzyszącą mu etykietą,

(2) Dla każdej z możliwych wymian, które można dokonać z pobranego elementu:

   - sprawdź, czy nie została osiągnięta konfiguracja otwierająca Sezam. Jeśli tak, to zakończ algorytm wypisując wartość etykiety badanej konfiguracji z początku kolejki, powiększoną o 1;
   - jeśli Sezamu nie da się otworzyć za pomocą otrzymanej konfiguracji, to sprawdź, czy któraś ze współrzędnych nie przekracza 100. Jeśli tak, to zignoruj tę konfigurację.
   - w przeciwnym razie wstaw do kolejki uzyskaną konfigurację, etykietując ją liczbą o jeden większą, niż etykieta, która towarzyszyła konfiguracji pobranej z kolejki.

Pokazanie w ścisły sposób, że faktycznie 100 jest właściwym ograniczeniem, nie jest proste. Wielu uczestników wpadło na inne, niepoprawne ograniczenia (do typowych należały $16, 40, 64$) i zapewne też byli przekonani, że ich ograniczenia są właściwe.

Tutaj od razu pewna dygresja. Niestety wśród testów nie znalazł się taki, jak opisany powyżej, który by wymagał ograniczenia wartości współrzędnej do 100. W związku z tym uczestnicy, którzy przyjęli niepoprawnie mniejsze ograniczenia, mogli uzyskać (i czasem uzyskiwali) maksymalne liczby punktów trochę niezasłużenie. Słusznie ci, którzy przyjmowali właściwe ograniczenia mieli później pretensje, że testy nie wychwyciły wyższości ich rozwiązania nad niezupełnie poprawnymi rozwiązaniami kolegów. Jest to pewna, niestety trudna do uniknięcia wada sprawdzania za pomocą systemu testów. Jestem przekonany, że i w przeszłości takie przypadki, choć niekoniecznie uświadomione, mogły się wydarzyć. Jakkolwiek organizatorzy Olimpiady starają się tak dobrać testy, aby wychwycić potencjalnie złe rozwiązania, jednak nie zawsze udaje się doścignąć inwencję zawodników w generowaniu niepoprawnych kodów. Tym razem niedopatrzenie było widoczne gołym okiem. Niewykluczone, że w przyszłych edycjach Olimipiady system punktowania i przygotowywania testów zostanie nieco zmodyfikowany, na przykład przez dodanie punktu umożliwiającego wprowadzenie dodatkowych testów po obejrzeniu rozwiązań zawodników.

Powstaje przy okazji tego rozwiązania następujące pytanie. Na ile rozwiązanie nieco przypadkowo prawidłowe jest godne najwyższej oceny? To jest trudny problem. Wyobraźmy sobie taką sytuację. Dany problem można rozwiązać za pomocą dwóch algorytmów. Jeden z nich, wolniejszy, posiada dowód poprawności. Drugi, oparty na jakimś fajnym pomyśle, działa szybciej, ale nie za bardzo wiadomo dlaczego, i ogólnie nigdy nie ma pewności, czy jest do końca poprawny. Profesjonaliści zdecydowanie wolą pierwsze rozwiązanie. Najczęściej praca z niepewnym algorytmem jest dość stresująca i, na dłuższą skalę, nieopłacalna.

Czy potrafimy zatem przedstawić algorytm rozwiązujący to zadanie w sposób pewny? Chodzi tu jeszcze o jedną rzecz: łatwość uogólnienia. Czy bowiem bylibyśmy w stanie podać równie łatwo odpowiednie ograniczenie, gdyby liczba rodzajów żetonów była większa, a początkowe konfiguracje ograniczone przez liczby większe niż 4? Czy to ograniczenie byłoby w ogóle użyteczne (co nam po nim, jeśli powiedzmy byłoby rzędu 10^{12} zamiast 10^2)?

Okazuje się, że rozwiązać nasz problem można za pomocą algorytmu jakby dualnego do podanego powyżej. Zastosujemy również przechodzenie grafu wszerz, ale tym razem będziemy rozwiązania szukali od końca. Wyjdziemy zatem od Sezamu, jako konfiguracji początkowej i będziemy zastanawiali się, jakie konfiguracje mogą doprowadzić do Sezamu, albo do konfiguracji od niego większej.

W tym celu, mając w ręku konfigurację v pobraną z kolejki, będziemy badali, ile co najmniej trzeba mieć żetonów każdego rodzaju, aby stosując daną regułę zamiany otrzymać konfigurację równą v, lub od niej większą. Zilustrujmy to dwoma przykładami. Jeżeli $v = (4, 4, 4)$, a badana reguła zamiany jest postaci $(2, 2, 2) \rightarrow (2, 1, 3)$, to najmniejsza liczba żetonów, którą trzeba mieć aby otrzymać v lub więcej niż v, wynosi odpowiednio $(4, 5, 3)$. Jeżeli natomiast inna reguła zamiany jest postaci, $(6, 2, 4) \rightarrow (5, 3, 3)$, to taka najmniejsza konfiguracja jest równa $(6, 3, 5)$. Ogólny wzór wyliczający i-tą współrzędną takiej konfiguracji, to $\max(wez[i], daj[i] - wez[i] + v[i])$, gdzie $daj[i]$ oraz $wez[i]$, to odpowiednio liczby żetonów, które trzeba dać i które się dostanie stosując i-tą regułę. Wyliczenie tych wartości dokonuje się w procedurze *oktant* (w programie ALI.PAS) w czasie generowania nowych elementów kolejki.

Jeżeli tak otrzymana konfiguracja będzie mniejsza, niż początkowa konfiguracja Alibaby, to kończymy algorytm. Jeżeli nie, to zanim wstawimy ją do kolejki, sprawdzamy, czy nie jest ona gorsza od którejś z już przebadanych konfiguracji (w tym celu w programie tworzymy listę *osiag*, w której pamiętamy wszystkie osiągnięte konfiguracje). Gorsza, to znaczy mniejsza lub równa na wszystkich współrzędnych od konfiguracji już przebadanej, która ze względu na właściwości algorytmu przechodzenia grafu wszerz jest na domiar bliższa Sezamowi. Nie dopuszczamy zatem takiej sytuacji, że w kolejce znalazłyby się dwie porównywalne ze sobą konfiguracje (tzn. takie, że wszystkie współrzędne jednej z nich byłyby nie mniejsze, niż wszystkie współrzędne drugiej).

Powstaje pytanie, w czym podany algorytm różni się od poprzednio omawianego. Na pierwszy rzut oka może się wydawać, że struktury, z którymi mamy do czynienia, czyli kolejka oraz lista napotkanych konfiguracji, mogą potencjalnie urosnąć do nieograniczonych rozmiarów. Udowodnimy teraz, że tak się stać nie może: liczba nieporównywalnych ze sobą punktów o współrzędnych naturalnych nie może być nieskończona. Fakt ten znany w różnych wersjach pod nazwą lematu Dicksona jest bardzo użytecznym wynikiem. Podamy tu jego intuicyjny dowód. Uściślijmy na początku parę pojęć.

Rozważamy przestrzeń wektorów $\mathbb{N}^n$, czyli $n-$wymiarowych wektorów o współrzędnych całkowitych, nieujemnych. Dla $v \in \mathbb{N}^n$ przez $v[i]$ będziemy oznaczali i-tą współrzędną wektora v dla $i = 1, \cdots, n$. Dwa wektory $v, w \in \mathbb{N}^n$ nazwiemy **nieporównywalnymi**, jeśli istnieją takie $1 \leqslant i, j \leqslant n$, że $v[i] < w[i]$ oraz $v[j] > w[j]$. Czyli ze względu na jedną ze współrzędnych „lepszy” jest jeden wektor, a ze względu na inną — drugi. W przeciwnym razie powiemy, że wektory są **porównywalne**, i wtedy możemy je uszeregować według wielkości, bowiem wszystkie wartości współrzędnych jednego z nich nie mogą przekroczyć odpowiadających im wartości drugiego. W takim przypadku, podobnie jak dla liczb naturalnych, możemy mówić o relacjach mniejszości, niewiększości, niemniejszości i większości między dwoma wektorami.

Twierdzenie 1: (Lemat Dicksona) W każdym nieskończonym ciągu wektorów $\mathbb{N}^n$ istnieje nieskończony podciąg niemalejący.

Dowód: Indukcja ze względu na wymiar n.

Podstawa indukcji: $n = 1$. Jednowymiarowe wektory $\mathbb{N}^1$ można traktować, jak liczby naturalne. Załóżmy przeciwnie, że nie istnieje w pewnym nieskończonym ciągu $a_1, a_2, \ldots$ liczb naturalnych podciąg niemalejący. Musi wtedy istnieć element a_i taki, że dla wszystkich $j > i : a_j < a_i$. Gdyby takiego elementu nie było, to mielibyśmy sposób konstrukcji nieskończonego niemalejącego podciągu: jako pierwszy wzięlibyśmy element a_1, jako drugi — element nie mniejszy od niego o większym indeksie, jako trzeci — element nie mniejszy od drugiego o większym indeksie, niż drugi, itd. Skoro zatem żaden z elementów ciągu nie przekracza a_i, to zbiór wszystkich wartości występujących w ciągu jest skończony, a ponieważ sam ciąg jest nieskończony, to przynajmniej jedna z wartości musi powtórzyć się nieskończenie wiele razy. Podciąg tych wartości jest nieskończonym podciągiem niemalejącym, wbrew założeniu. Sprzeczność. Skorzystaliśmy tu z nieskończonego wariantu zasady szufladkowej Dirichleta.

Krok indukcyjny: Załóżmy prawdziwość twierdzenia dla wszystkich wymiarów aż do $n - 1$ włącznie, $n > 1$. Wybierzmy z danego ciągu $a_1, a_2, \ldots$ nieskończony podciąg $a_{i_1}, a_{i_2}, \ldots$ niemalejący ze względu na początkowe $n-1$ wymiarów. Na mocy założenia indukcyjnego taki podciąg istnieje. Ciąg liczb naturalnych $a_{i_1}[n], a_{i_2}[n], \ldots$ zawiera nieskończony podciąg niemalejący $a_{i_{j_1}}[n], a_{i_{j_2}}[n], \ldots$. Rozważmy nieskończony podciąg wektorów $a_{i_{j_1}}, a_{i_{j_2}}, \ldots$. Ze względu na pierwszych $n - 1$ współrzędnych jest on niemalejący, ale również ze względu na ostatnią współrzędną jest niemalejący. Stąd jest niemalejący ze względu na wszystkie współrzędne, co kończy dowód. □

Z przedstawionego powyżej lematu wynika

Wniosek 2: Nie istnieje nieskończony zbiór nieporównywalnych wektorów z $\mathbb{N}^n$.

Gdyby taki zbiór miał istnieć, to ustawilibyśmy elementy tego zbioru w nieskończony ciąg, w którym znaleźlibyśmy nieskończony podciąg (a więc znaleźlibyśmy i podzbiór) niemalejący.

Z powyższego faktu wynika poprawność konstrukcji algorytmu. Nie ma niebezpieczeństwa, że algorytm nasz mógłby się zapętlić. Zbiór wektorów, które mają szansę znaleźć się w liście *osiag* jest skończony. Algorytm nasz musi się więc zakończyć w skończonym czasie. Z każdym krokiem algorytmu lista musiałaby się bowiem albo powiększyć, albo któryś z jej elementów musiałby ulec zmniejszeniu. Gdyby algorytm działał w nieskończoność, wówczas pierwsza z tych rzeczy byłaby wykluczona na mocy udowodnionego właśnie faktu, a druga na mocy tego, że skończonych wektorów nie jesteśmy w stanie nieskończenie wiele razy zmniejszać.

Zauważmy, że dowód poprawności (a tak naprawdę własności stopu) algorytmu jest niekonstruktywny. Nie wiemy, jakie jest ograniczenie czasu jego działania, nie wiemy, jak będą wyglądały budowane zbiory, wiemy tylko, że nie mogą urosnąć one do nieskończonych rozmiarów.

Oczywiście czasami, z pewnego punktu widzenia, ze względu na dużą złożoność algorytmu, możemy mówić o jego praktycznej nieskończoności. Gdyby na przykład okazało się, że dane początkowe wymagają, żeby przechowywany zbiór miał powiedzmy 2^{100} elementów, to praktycznie byłby to zbiór nieskończony. I to, czy sam algorytm jest nieskończony, czy też jest skończony, ale niewyobrażalnie duży, nie miałoby znaczenia. Po prostu i w jednym i w drugim przypadku byłby nie do użycia. W naszym zadaniu, przyjęte dane powodują, że niewątpliwie zbiór ten nie będzie zbyt duży. Oszacowanie 100^3 (na każdej współrzędnej co najwyżej 100) jest bardzo grubym oszacowaniem i w rzeczywistości lista *osiag* nie przekracza nigdy kilkuset elementów.

TESTY

Do sprawdzenia rozwiązań zawodników użyto testów zebranych w sześciu plikach ALI0.IN–ALI5.IN

- ALI0.IN — 2 testy z treści zadania,
- ALI1.IN — 4 proste testy testy niskowymiarowe: z jedną lub dwiema współrzędnymi aktywnymi;
- ALI2.IN — 2 niewielkie testy: pierwszy na „nie”, drugi generujący sporą liczbę wymian;
- ALI3.IN — 2 testy, w których długość ścieżki była dość duża (ponad 100).
- ALI4.IN — 2 testy badające wydajność algorytmu. Oba generowały duże grafy osiągalności (przy opisanym algorytmie działającym wstecz), przy czym pierwszy z nich nie miał rozwiązania, a drugi — miał;
- ALI5.IN — 2 spore testy z nieco większą liczbą reguł wymiany.

Wojciech Rytter
treść zadania, opracowanie

Marcin Mucha
program wzorcowy

Kajaki

Organizując spływ, wypożyczamy na przystani kajaki. Wszystkie kajaki są jednakowe. W jednym kajaku mogą popłynąć co najwyżej dwie osoby, a suma ich wag nie może przekroczyć ustalonego maksymalnego obciążenia. Aby zapłacić jak najmniej, szukamy sposobu rozmieszczenia wszystkich uczestników spływu w minimalnej liczbie kajaków.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego KAJ.IN maksymalne obciążenie kajaka, liczbę uczestników spływu i wagę każdego z nich,*
- *oblicza minimalną liczbę kajaków, jakie należy wynająć, aby rozmieścić w nich wszystkich uczestników spływu w zgodzie z przepisami,*
- *zapisuje wynik w pliku tekstowym KAJ.OUT.*

Wejście

W pierwszym wierszu pliku tekstowego KAJ.IN jest zapisana jedna liczba całkowita w, spełniająca nierówności $80 \leqslant w \leqslant 200$. Jest to maksymalne obciążenie kajaka.
W drugim wierszu jest zapisana jedna liczba całkowita n, spełniająca nierówności $1 \leqslant n \leqslant 30000$. Jest to liczba uczestników spływu.
W każdym z kolejnych n wierszy jest zapisana jedna liczba całkowita z przedziału $[5..w]$. Jest to waga jednego uczestnika spływu.

Wyjście

W pierwszym wierszu pliku tekstowego KAJ.OUT należy zapisać jedną liczbę całkowitą — minimalną liczbę kajaków, jakie trzeba wynająć.

Przykład

Dla pliku tekstowego KAJ.IN:

```
100
9
90
20
20
30
50
60
70
80
90
```

poprawnym rozwiązaniem jest plik tekstowy KAJ.OUT

```
6
```

Twój program powinien szukać pliku KAJ.IN w katalogu bieżącym i tworzyć plik KAJ.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę KAJ.??? gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonalnej powinien być zapisany w pliku KAJ.EXE.

ROZWIĄZANIE

Zadanie jest bardzo proste. Do jego rozwiązania możemy zastosować metodę **przeciwnych wag**: najlżejszą osobę wsadzamy do pierwszego kajaka z jak najcięższą osobą, która się z nią zmieści (tzn. nie zostanie przekroczone dopuszczalne obciążenie). Następnie kontynuujemy postępowanie dla grupy pozostałych osób. Na końcu zostaną pojedyncze „grubasy”, których wsadzamy do pojedynczych kajaków.

Powyższe rozwiązanie można zaimplementować na różne sposoby. Najprościej uporządkować osoby niemalejąco względem wag, a następnie poszukiwać pasażerów kajaków wybierając lekkich kandydatów z lewej strony, a ich partnerów od strony prawej.

Można podać jeszcze prostszy algorytm. W jednym kroku wybieramy **jakąkolwiek** wolną osobę, po czym wsadzamy ją do kajaka razem z jak najcięższą jeszcze wolną osobą, która się z nią mieści. Poprzednie rozwiązanie jest szczególnym przypadkiem tej ogólnej strategii.

Dlaczego taka strategia jest poprawna? Załóżmy, że na początku bierzemy osobę i. Jeśli i nie mieści się do kajaka z nikim innym, to w każdym optymalnym układzie i płynie samotnie, tak więc nasz optymalny układ dla pozostałych osób zagwarantuje optymalność rozwiązania globalnego. Jeśli i wsadzimy do kajaka z osobą j, a w pewnym optymalnym układzie i płynie z j', to w tym układzie możemy zamienić j z j', ponieważ waga j' jest nie większa od wagi j.

IMPLEMENTACJA

Efektywność implementacji przedstawionego algorytmu zależy od doboru metody sortowania, ponieważ dobieranie osób do kajaków wykonuje się w czasie liniowym. W rozwiązaniu wzorcowym zastosowano sortowanie kubełkowe. Poza tym nie kompletowano pojedynczych kajaków, ale całe grupy. Jeśli przykładowo mamy dziesięciu uczestników spływu o wadze 60 i pięciu o wadze 90, a maksymalne obciążenie kajaka wynosi 150, to możemy od razu obsadzić pięć kajaków.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 10 testów (KAJ0.IN–KAJ9.IN). Test KAJ0.IN jest przykładem z treści zadania. Testy KAJ1.IN–KAJ4.IN są prostymi testami sprawdzającymi poprawność implementacji algorytmu. Testy KAJ5.IN–KAJ9.IN są dużymi (po 30 tys. uczestników spływu) testami wydajności badającymi głównie efektywność zastosowanej metody sortowania.

Wojciech Rytter
treść zadania, opracowanie

Krzysztof Sobusiak
program wzorcowy

Liczba zbiorów n-k-specjalnych

Mówimy, że zbiór liczb naturalnych X jest n-k-specjalny, gdy:

- *każdy element $x \in X$ jest liczbą naturalną spełniającą nierówność $1 \leqslant x \leqslant n$,*
- *suma wszystkich liczb należących do X jest większa od k,*
- *X nie zawiera żadnej pary kolejnych liczb naturalnych.*

ZADANIE

Napisz program, który:

- *wczytuje z pliku tekstowego LIC.IN dwie liczby naturalne n oraz k,*
- *znajduje liczbę wszystkich zbiorów n-k-specjalnych,*
- *zapisuje wynik w pliku tekstowym LIC.OUT.*

WEJŚCIE

W pierwszym wierszu pliku tekstowego LIC.IN są zapisane dwie liczby naturalne n oraz k oddzielone pojedynczym odstępem, gdzie $1 \leqslant n \leqslant 100$, $0 \leqslant k \leqslant 400$.

WYJŚCIE

W pierwszym wierszu pliku tekstowego LIC.OUT należy zapisać jedną liczbę całkowitą nieujemną — liczbę wszystkich zbiorów n-k-specjalnych dla danych n oraz k.

PRZYKŁAD

Dla pliku tekstowego LIC.IN:

```
5 6
```

poprawnym rozwiązaniem jest plik tekstowy LIC.OUT

```
3
```

Twój program powinien szukać pliku LIC.IN w katalogu bieżącym i tworzyć plik LIC.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę LIC.??? gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonalnej powinien być zapisany w pliku LIC.EXE.

ROZWIĄZANIE

Rozpoczniemy od wprowadzenia niezbędnych definicji. W tym opracowaniu pisząc „zbiór” mamy na myśli zawsze podzbiór liczb całkowitych. **Wagą** zbioru nazywamy sumę jego elementów. Wagą zbioru pustego jest 0. Powiemy, że zbiór jest **specjalny**, jeżeli nie zawiera dwóch kolejnych liczb całkowitych. Zbiór jest j**-lekki**, gdzie j jest nieujemną liczbą całkowitą, jeżeli jego waga nie przekracza j.

Przez $S(i)$ będziemy oznaczali liczbę wszystkich podzbiorów specjalnych zbioru $\{1, 2, \ldots, i\}$ dla $i = 0, 1, \ldots$, natomiast $ILE(i, j)$ będzie oznaczało liczbę wszystkich j-lekkich podzbiorów zbioru $\{1, 2, \ldots, i\}$. Dodatkowo przyjmijmy $ILE(i, j) = 0$ dla $j < 0$ oraz $ILE(i, j) = 1$ dla $i < 0$ i $j \geqslant 0$.

Rys. 1 Tablice ILE i S oraz przykładowy schemat liczenia $ILE(3,5)$. Jeśli $n = 5$, $k = 5$ (dane z przykładu w treści zadania) to wynikiem jest $S(5) - ILE(5,5) = 5$.

	1	2	3	4	5
1	2	2	2	2	2
2	2	2	3	3	3
3	2	3	4	4	4
4	2	3	5	6	6
5	2	3	5	7	8

ILE

1	2
2	3
3	5
4	8
5	13

S

Zadanie rozwiązuje się metodą tablicowania (lub inaczej programowania dynamicznego), wprowadzając pomocniczą tablicę dla zapamiętywania wyników pośrednich. Istota zadania polega na odpowiednim wydefiniowaniu dodatkowych pomocniczych danych oraz zależności rekurencyjnych. Zadanie jest w pewnym sensie podobne do liczenia liczb Fibonnaciego. Łatwo zauważyć następujący fakt:

Fakt 1: Liczba $S(i)$ podzbiorów specjalnych zbioru $\{1, 2, \ldots, i\}$ jest $(i+1)$-szą liczbą Fibonacciego, tzn. $S(0) = 1$, $S(1) = 2$, $S(i) = S(i-1) + S(i-2)$ dla $i > 1$.*

Nasz algorytm będzie wykonywać kolejne iteracje dla $i = 1, 2, \ldots n$. W i-tej iteracji policzymy $ILE(i,j)$ dla $1 \leqslant j \leqslant k$. Po zakończeniu ostatniej iteracji obliczymy wynik odejmując od odpowiedniej liczby Fibonacciego $S(n)$ liczbę $ILE(n,k)$, (zob. rys. 1). W i-tej iteracji obliczamy i-ty wiersz tablicy ILE korzystając z rekurencji:

$$ILE(i,j) = ILE(i-1,j) + ILE(i-2,j-i)$$

Rekurencja ta oznacza, że każdy j-lekki podzbiór zbioru $\{1, 2, \ldots, i\}$, albo zawiera i (wtedy po usunięciu i jest $(j-i)$-lekkim podzbiorem zbioru $\{1, .., i-2\}$), albo nie zawiera i (wtedy jest j-lekkim podzbiorem zbioru $\{1, .., i-1\}$).

UWAGA: W danej iteracji potrzebne nam są tylko trzy wiersze tablicy ILE: aktualnie obliczany wiersz i poprzednie dwa wiersze. Spostrzeżenie to istotnie zmniejsza zapotrzebowanie programu na pamięć.

Ze względu na możliwość wystąpienia dużych liczb naturalnych należy zaimplementować własną arytmetykę z operacjami $+$ i $-$. Złożoność czasowa algorytmu jest rzędu $n \cdot k$.

Można wyróżnić trzy klasy rozwiązań, które prowadziły do programów niepoprawnych lub nieefektywnych:

- rozwiązania starające się trzymać w pamięci całą tablicę ILE, co przy dużych danych wejściowych wykraczało poza dostępną pamięć operacyjną;
- rozwiązania rekurencyjne o złożoności wykładniczej;
- rozwiązania bez zaimplementowanej arytmetyki dużych liczb.

* Równanie rekurencyjne wynika z dodania liczby podzbiorów, do których nie należy element i, tj. $S(i-1)$, do liczby podzbiorów, do których ten element należy, tj. $S(i-2)$.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 11 testów LIC.0IN–LIC10.IN. Test LIC0.IN był testem z treści zadania. Celem testów LIC1.IN–LIC6.IN było sprawdzenie poprawności rozwiązań zawodników dla danych o stosunkowo niewielkich rozmiarach. Testy LIC7.IN–LIC10.IN to testy wydajności, sprawdzające zarówno szybkość działania programów, jak i zużycie pamięci.

Wojciech Guzicki
treść zadania, opracowanie

Tomasz Błaszczyk
program wzorcowy

Rezerwacja sal wykładowych

Rozporządzamy jedną salą wykładową. Wykładowcy, którzy chcą korzystać z sali, składają zamówienia określając czas rozpoczęcia i zakończenia wykładu. Układamy plan wykorzystania sali akceptując pewne wykłady i odrzucając inne, tak aby czas wykorzystania sali był jak najdłuższy. Zakładamy, że w momencie zakończenia jednego wykładu może się rozpocząć następny wykład.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego REZ.IN zamówienia wykładowców,*
- *oblicza maksymalny czas wykorzystania sali (przy odpowiednio ułożonym planie wykładów),*
- *zapisuje wynik w pliku tekstowym REZ.OUT.*

Wejście

W pierwszym wierszu pliku tekstowego REZ.IN jest zapisana jedna liczba całkowita dodatnia $n \leqslant 10000$. Jest to liczba zamówień.
W każdym z kolejnych n wierszy są zapisane dwie liczby całkowite p oraz k, oddzielone pojedynczym odstępem spełniające nierówności $0 \leqslant p \leqslant k \leqslant 30000$. Jest to zamówienie na salę wykładową w przedziale czasu od p do k.

Wyjście

W pierwszym wierszu pliku tekstowego REZ.OUT należy zapisać maksymalny czas wykorzystania sali.

Przykład

Dla pliku tekstowego REZ.IN:

```
12
1 2
3 5
0 4
6 8
7 13
4 6
9 10
9 12
11 14
15 19
14 16
18 20
```

poprawnym rozwiązaniem jest plik tekstowy REZ.OUT

```
16
```

Twój program powinien szukać pliku REZ.IN w katalogu bieżącym i tworzyć plik REZ.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę REZ.??? gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonalnej powinien być zapisany w pliku REZ.EXE.

UZUPEŁNIENIE TREŚCI

Podczas trwania zawodów treść zadania uzupełniono następującym stwierdzeniem:

> *Zamówienie w postaci pary liczb całkowitych p, k oznacza, że wykładowca potrzebuje sali w czasie reprezentowanym na osi czasu przez odcinek otwarty*

ROZWIĄZANIE

Zadanie to jest modyfikacją jednego z zadań z książki [10]. Oryginalne zadanie polegało na takim przydziale sal wykładowcom, by zadowolić jak największą liczbę zainteresowanych. W takim sformułowaniu zadanie można było rozwiązać za pomocą algorytmu zachłannego.

Obecne zadanie można rozwiązać za pomocą programowania dynamicznego. Dane wczytujemy do tablicy T typu

$$\textbf{array}[1..MAXN] \textbf{ of } \mathit{TRekord}$$

gdzie typ *TRekord* jest zdefiniowany w następujący sposób:

$$\mathit{TRekord} = \textbf{record } \mathit{Pocz}, \mathit{Kon} : \textbf{Word end};$$

Będziemy także używać tablicy *Czas* typu

$$\textbf{array}[\mathbf{0}..MAXN] \textbf{ of Word};$$

Oczywiście $T[i].\mathit{Pocz}$ oznacza czas rozpoczęcia i-tego wykładu, a $T[i].\mathit{Kon}$ czas zakończenia tego wykładu. Następnie tablicę tę sortujemy niemalejąco względem czasów zakończenia wykładów i zaczynamy kolejno wyznaczać liczby $\mathit{Czas}[i]$. Liczba $\mathit{Czas}[i]$ oznacza najlepszy możliwy czas wykorzystania sali przez wykładowców od pierwszego do i-tego (numery wykładowców po posortowaniu tablicy T). Oczywiście

$$\mathit{Czas}[1] = T[1].\mathit{Kon} - T[1].\mathit{Pocz}$$

Dla uproszczenia przyjmujemy także $\mathit{Czas}[0] = 0$. Dla $i > 1$ porównujemy dwie liczby. Pierwszą jest $t1 = \mathit{Czas}[i-1]$, oznaczająca optymalny czas wykorzystania sali w przypadku, gdybyśmy zrezygnowali z przydzielenia sali i-temu wykładowcy. Drugą, w programie oznaczoną przez *t2*, otrzymamy dodając czas trwania i-tego wykładu do najlepszego możliwego czasu wykorzystania sali przez wykłady kończące się nie później niż w chwili $T[i].\mathit{Pocz}$. Ten czas znajdujemy w następujący sposób. Zauważamy najpierw, że liczby w tablicy *Czas* są uporządkowane niemalejąco. Niemalejąco są również uporządkowane rekordy w tablicy T względem pola *Kon*. Możemy więc zastosować wyszukiwanie binarne. Funkcja *znajdź_indeks* znajduje najwcześniej występujący wykład kończący się później niż w chwili $T[i].\mathit{pocz}$. Jeśli jest to pierwszy wykład, to żaden nie mógł rozpocząć się przed i-tym wykładem i dotychczasowy najlepszy czas wynosi 0 (zauważmy, że jest to $\mathit{Czas}[0]$). Jeśli nie jest to pierwszy wykład, to w poprzednim miejscu w tablicy *Czas* znajduje się szukany najlepszy dotychczasowy czas. W obu przypadkach jest to $\mathit{Czas}[j-1]$, gdzie j jest liczbą znalezioną za pomocą funkcji *znajdź_indeks*, a więc najmniejszą liczbą taką, że $T[j].\mathit{Kon} > T[i].\mathit{Pocz}$. Teraz jako $\mathit{Czas}[i]$ przyjmujemy większą z liczb *t1* i *t2*. Liczba $\mathit{Czas}[n]$ jest szukanym najlepszym czasem wykorzystania sali.

Nietrudno dowieść przez indukcję względem i, że na i-tym miejscu w tablicy *Czas* rzeczywiście znajduje się optymalny czas wykorzystania sali przez wykładowców od pierwszego do i-tego.

Implementacja tego algorytmu jest już prosta. Najpierw sortujemy wczytane dane. W programie wzorcowym zastosowano sortowanie z użyciem kopca (zob. [7]). Działa ono w miejscu (tzn. sortuje elementy tablicy nie korzystając z dodatkowej pamięci — używa tylko kilku dodatkowych zmiennych pomocniczych) i przy tym dość szybko. Czas działania $O(n \log n)$ nie zależy też w istotny sposób od danych, w przeciwieństwie do sortowania szybkiego (*quicksort*). W przypadku, gdy mamy do czynienia z tablicą posortowaną i w sortowaniu szybkim wybieramy jako element dzielący pierwszy element, to sortowanie szybkie będzie wymagało czasu kwadratowego. Oczywiście implementacje sortowania szybkiego wybierające element dzielący staranniej (np. losowo) też będą działały szybko.

Następnie wypełniamy tablicę *Czas* w sposób opisany powyżej. Wreszcie zapisujemy w pliku wyjściowym liczbę $Czas[n]$.

Złożoność obliczeniowa podanego algorytmu wynosi $O(n \log n)$. Sortowanie wstępne wykonujemy w tym właśnie czasie, a następnie n razy dokonujemy wyszukiwania binarnego w czasie $O(\log n)$.

Można rozwiązać to zadanie z pominięciem sortowania. Wtedy złożoność wyniesie $O(n^2)$. Można zastosować sortowanie szybkie. Przy niestarannym wyborze elementu dzielącego można też uzyskać czas rzędu n^2. Można też użyć algorytmu z sortowaniem, ale bez wyszukiwania binarnego (z wyszukiwaniem liniowym). Ten algorytm też ma złożoność $O(n^2)$.

Istnieje również niebezpieczeństwo zastosowania algorytmu zachłannego, służącego do rozwiązania zadania oryginalnego. Taki algorytm często da wynik błędny.

TESTY

Zastosowano 13 testów (REZ0.IN–REZ12.IN). Pierwszy (REZ0.IN) był testem z treści zadania. Drugi był testem złożonym z tylko jednego odcinka. Trzeci był posortowanym ciągiem 20 zamówień, z których każde zaczynało się w momencie zakończenia poprzedniego. Czwarty składał się z nieposortowanego ciągu dwóch serii zamówień o długości 600 i 601, przy czym jedna seria była przesunięta względem drugiej o 30 jednostek. Dwa następne testy miały wyeliminować rozwiązania korzystające z algorytmu zachłannego. Siedem następnych testów to testy składające się z dużej liczby odcinków różnie uporządkowanych. Dwa z tych testów (REZ10.IN–REZ11.IN) zawierały dane losowe.

Wojciech Guzicki
treść zadania, opracowanie

Marcin Engel
program wzorcowy

Trójkąty jednobarwne

W przestrzeni rozmieszczono n punktów w taki sposób, że żadne trzy z nich nie są współliniowe. Następnie każdą parę tych punktów połączono odcinkiem i każdy odcinek pokolorowano na czarno albo na czerwono. Trójkątem jednobarwnym nazwiemy każdy trójkąt mający wszystkie trzy boki tego samego koloru. Mamy daną listę wszystkich czerwonych odcinków. Chcemy znaleźć liczbę wszystkich trójkątów jednobarwnych.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego TRO.IN: liczbę punktów, liczbę odcinków czerwonych oraz ich listę,*
- *znajduje liczbę trójkątów jednobarwnych,*
- *zapisuje wynik w pliku tekstowym TRO.OUT.*

Wejście

W pierwszym wierszu pliku tekstowego TRO.IN jest zapisana jedna liczba całkowita n spełniająca nierówności $3 \leqslant n \leqslant 1000$. Jest to liczba punktów.
W drugim wierszu pliku tekstowego TRO.IN jest zapisana jedna liczba całkowita m spełniająca nierówności $0 \leqslant m \leqslant 250000$. Jest to liczba odcinków czerwonych.
W każdym z kolejnych m wierszy są zapisane dwie liczby całkowite p oraz k, oddzielone pojedynczym odstępem i spełniające nierówność: $1 \leqslant p \leqslant k \leqslant n$. Są to numery wierzchołków będących końcami kolejnego odcinka czerwonego.

Wyjście

W pierwszym wierszu pliku tekstowego TRO.OUT należy zapisać jedną liczbę całkowitą — liczbę trójkątów jednobarwnych.

Przykład

Dla pliku tekstowego TRO.IN:

```
6
9
1 2
2 3
2 5
1 4
1 6
3 4
4 5
5 6
3 6
```

poprawnym rozwiązaniem jest plik tekstowy TRO.OUT

```
2
```

Twój program powinien szukać pliku TRO.IN w katalogu bieżącym i tworzyć plik TRO.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę TRO.??? gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonalnej powinien być zapisany w pliku TRO.EXE.

ROZWIĄZANIE

Rozwiązanie narzucające się, polegające na przeszukaniu wszystkich trójek punktów, jest zbyt czasochłonne (czas rzędu n^3). Pomysł na szybszy algorytm polega na zliczaniu trójkątów różnobarwnych. Jeśli z wierzchołka o numerze k wychodzi $d(k)$ odcinków czerwonych i $n-1-d(k)$ odcinków czarnych, to liczba trójkątów różnobarwnych, o różnokolorowych bokach i wspólnym końcu w punkcie k, jest równa $d(k)\cdot(n-1-d(k))$. Liczba trójkątów różnobarwnych jest wtedy równa

$$S = \frac{1}{2}\cdot\sum_{k=1}^{n} d(k)\cdot(n-1-d(k))$$

Oczywiście liczby $d(k)$ wyznaczamy w czasie wczytywania danych o odcinkach czerwonych. Czas działania algorytmu będzie więc liniowy względem liczby danych (a więc co najwyżej $O(n^2)$). Liczbę trójkątów jednobarwnych wyznaczamy odejmując otrzymaną sumę od liczby wszystkich trójkątów:

$$J = \binom{n}{3} - S.$$

Wszystkie liczby mieszczą się w zakresie typu *longint*. Program wzorcowy znajduje się w pliku `TRO.PAS`, program działający w czasie rzędu n^3 znajduje się w pliku `TRO1.PAS`.

TESTY

Pierwszy test (TRO0.IN) był testem z treści zadania. Drugi, dla $n = 4$, nie miał odcinków czerwonych. Pięć następnych testów to testy losowe dla liczb n równych odpowiednio 10, 30, 100, 300, 600.

Grzegorz Jakacki
program wzorcowy, treść zadania, opracowanie

Wiarygodność świadków

Sąd dysponuje zeznaniami świadków oznaczonych kolejno liczbami naturalnymi $1, 2, \ldots, n$. Każde zeznanie jest stwierdzeniem postaci: „świadek i zgadza się ze świadkiem j" albo „świadek i nie zgadza się ze świadkiem j".
Jeśli świadek i zgadza się ze świadkiem j, oznacza to, że świadek i zgadza się też ze wszystkimi świadkami, z którymi zgadza się się świadek j oraz nie zgadza się ze wszystkimi świadkami, z którymi nie zgadza się świadek j.
Mówimy, że świadek nie jest wiarygodny*, jeśli z zeznań świadków, będących w posiadaniu sądu wynika, że jednocześnie zgadza się on i nie zgadza z pewnym dowolnym świadkiem.*

Zadanie

Ułóż program, który:

- *wczytuje z pliku tekstowego WIA.IN liczbę świadków i ich zeznania,*
- *znajduje wszystkich świadków, którzy nie są wiarygodni,*
- *zapisuje wynik w pliku tekstowym WIA.OUT*

Wejście

W pierwszym wierszu pliku tekstowego WIA.IN jest zapisana liczba całkowita n, spełniająca nierówność $1 \leqslant n \leqslant 3000$. Jest to liczba świadków.
W drugim wierszu pliku tekstowego WIA.IN jest zapisana jedna liczba całkowita m, spełniająca nierówność $0 \leqslant m \leqslant 8000$. Jest to liczba zeznań.
W każdym z kolejnych m wierszy jest zapisana para liczb całkowitych i j, oddzielonych pojedynczym odstępem, gdzie $1 \leqslant i \leqslant n$, $1 \leqslant |j| \leqslant n$. Jeśli liczba j jest dodatnia — jest to zapis zeznania: „świadek i zgadza się ze świadkiem j". Jeśli jest to liczba ujemna – jest to zapis zeznania: „świadek i nie zgadza się ze świadkiem $-j$".

Wyjście

W pliku tekstowym WIA.OUT należy zapisać:

- *jedno słowo NIKT, jeśli na podstawie zeznań nie można znaleźć żadnego świadka, który nie jest wiarygodny,*
- *albo — w porządku rosnącym — numery świadków, którzy nie są wiarygodni.*

Przykład

Dla pliku tekstowego WIA.IN:

```
6
12
1 3
1 6
2 -1
3 4
4 1
4 -2
4 5
5 -1
5 -3
5 2
6 5
6 4
```

Poprawnym rozwiązaniem jest plik tekstowy WIA.OUT:

```
1
3
4
6
```

UZUPEŁNIENIE TREŚCI

Podczas zawodów treść uzupełniono następującym stwierdzeniem:

Przyjmuje się przez domniemanie, że każdy świadek stwierdza: „Zgadzam się ze sobą”.

ROZWIĄZANIE

Zestaw zeznań świadków będziemy traktować jako tzw. graf skierowany (digraf) z etykietowanymi krawędziami. Wierzchołki grafu będziemy utożsamiać ze świadkami, natomiast jego krawędzie z zeznaniami. Krawędź od wierzchołka a do wierzchołka b ma oznaczać, iż świadek reprezentowany przez wierzchołek a wypowiada się o świadku reprezentowanym przez wierzchołek b. Typ wypowiedzi („zgadzam się” lub „nie zgadzam się”) będzie reprezentowany przez etykiety krawędzi. Etykietami będą słowa „tak” i „nie” jako skróty zeznań „tak, zgadzam się” i „nie, nie zgadzam się”.

Krawędź od wierzchołka a do wierzchołka b etykietowaną „tak” lub „nie” będziemy oznaczać odpowiednio $a \xrightarrow{T} b$ i $a \xrightarrow{N} b$. Ścieżkę (być może pustą) od wierzchołka a do wierzchołka b złożoną tylko z krawędzi typu „tak” lub „nie” będziemy oznaczać odpowiednio $a \xrightarrow{T}{}^{*} b$ i $a \xrightarrow{N}{}^{*} b$ (zakładamy, że dla każdego wierzchołka a istnieje ścieżka $a \xrightarrow{T}{}^{*} a$, bo świadek zgadza się z samym sobą). Gdy istnieje ścieżka $a \xrightarrow{T}{}^{*} b$, to będziemy mówić, że a **popiera** b.

Niech a będzie pewnym wierzchołkiem grafu. Zauważmy, że:

Obserwacja 1: Świadek a zgadza się ze świadkiem b wtedy i tylko wtedy, gdy istnieje ścieżka $a \xrightarrow{T}{}^{*} b$.

O świadku możemy stwierdzić, że nie jest wiarygodny, jeżeli jednocześnie zgadza się i nie zgadza z pewnym dowolnym świadkiem.

Jeżeli a zgadza się b, to nie zgadza się również ze świadkami, z którymi nie zgadza się b. Stąd otrzymujemy:

Obserwacja 2: Świadek a nie zgadza się ze świadkiem c jeżeli istnieje świadek b taki, że istnieje ścieżka $a \xrightarrow{T}{}^{*} b \xrightarrow{N} c$

Zauważmy, że jeżeli świadek a osobiście nie zgadza się ze świadkiem c, to obserwacja też pozostaje prawdziwa (bo świadkiem b jest w takim przypadku sam a).

Świadek nie jest wiarygodny, jeżeli jednocześnie zgadza się z jakimś świadkiem i z nim się nie zgadza. Zatem:

Obserwacja 3: Świadek a nie jest wiarygodny, jeżeli istnieją świadkowie b i c tacy, że istnieją ścieżki $a \xrightarrow{T}{}^{*} b \xrightarrow{N} c$ oraz $a \xrightarrow{T}{}^{*} c$.

W powyższej obserwacji świadek c jest tym świadkiem, z którym a jednocześnie zgadza się i nie zgadza.

Na tym etapie rozważań jesteśmy gotowi sformułować algorytm będący rozwiązaniem zadania. Dla każdego świadka a będziemy wyznaczać zbiór świadków, z którymi się on zgadza

$$ZgadzaSię(a) = \{x : a \xrightarrow{T}{}^{*} x\}$$

Świadka a należy uznać za niewiarygodnego, jeżeli pomiędzy elementami $ZgadzaSię(a)$ istnieje krawędź $\xrightarrow{N}$. Taka sytuacja oznacza, że wśród świadków, z którymi zgadza się a istnieją świadkowie b i c, tacy, że $a \xrightarrow{T}{}^{*} b \xrightarrow{N} c$, co rzeczywiście każe uznać a za świadka niewiarygodnego. Z drugiej strony, jeżeli pomiędzy elementami zbioru *ZgadzaSię*(a) nie ma takiej krawędzi, to znaczy, że świadka a należy uznać za wiarygodnego.

Zbiór *ZgadzaSię*(a) będziemy konstruować przy pomocy metody znanej jako „przechodzenie grafu w głąb" (ang. *depth-first search, DFS*), pozwalającej m.in. wyznaczyć zbiór wierzchołków grafu, do których istnieją ścieżki z zadanego wierzchołka (tzw. zbiór wierzchołków osiągalnych z danego wierzchołka).

Schemat algorytmu DFS przedstawiono na rys. 1. Algorytm uruchomiony dla wierzchołka a zaznacza go jako odwiedzony, a następnie uruchamia się rekurencyjnie dla wszystkich wierzchołków, do których prowadzą krawędzie z wierzchołka a i które nie zostały odwiedzone wcześniej.

Po zakończeniu pracy algorytmu wierzchołki odwiedzone tworzą zbiór wierzchołków osiągalnych z wierzchołka startowego.

Czas pracy algorytmu DFS uruchomionego dla jednego wierzchołka grafu jest rzędu $O(m)$, gdzie m jest liczbą krawędzi. Nie trudno to zauważyć wziąwszy pod uwagę, że każdą krawędź rozważamy tylko jeden raz.

Głębokość rekursji nie przekroczy na pewno długości najdłuższej ścieżki, na której nie powtarzają się wierzchołki, więc jest nie większa niż n.

Przed wywołaniem procedury DFS należy wszystkim wierzchołkom grafu ustawić atrybut *odwiedzony* na **false**. Założono, że *sąsiedzi*(w) oznacza zbiór wszystkich wierzchołków, do których prowadzą krawędzie z w. Taki zbiór można zaimplementować np. jako listę lub tablicę (jeżeli wiadomo z góry, ilu maksymalnie sąsiadów może mieć wierzchołek). Pętla w wierszu (1) musi, przy konkretnej implementacji, realizować w jakiś sposób przeglądanie tego zbioru (kolejność przeglądania nie ma znaczenia).

Rys. 1 Schemat algorytmu DFS

```
procedure DFS(w : wierzcholek)
var
  v : wierzchołek;
begin
  w.odwiedzony := true
  for v ∈ sąsiedzi(w) do          (1)
    if not v.odwiedzony then DFS(v)
end
```

W algorytmie będącym rozwiązaniem zadania posłużymy się zmodyfikowaną wersją algorytmu DFS. Po pierwsze, do zbioru *sąsiedzi*(a) zaliczymy jedynie te wierzchołki, do których można dojść z a po krawędziach $\xrightarrow{T}$, dzięki czemu po zakończeniu pracy algorytmu uruchomionego dla wierzchołka a wartość atrybutu *odwiedzony* ustawiona na **true** wskaże elementy zbioru *ZgadzaSię*(a). W rozwiązaniu wzorcowym zamiast nazwy atrybutu *odwiedzony* użyto nazwy *popierany* (gdyż ustawienie tego atrybutu oznacza, że świadek reprezentowany przez dany wierzchołek jest *popierany* w swych zeznaniach przez świadka, którego wiarygodność aktualnie się sprawdza).

Wyznaczenie zbioru *ZgadzaSię*(a) to jednak jeszcze nie wszystko. Kolejną fazą sprawdzania wiarygodności świadka jest sprawdzenie, czy pomiędzy elementami tego zbioru nie istnieje krawędź $\xrightarrow{N}$. By usprawnić tę operację, przy okazji konstruowania zbioru *ZgadzaSię*(a) algorytm zapisuje numery jego elementów do pomocniczej tablicy (dokładniej: w procedurze DFS oprócz zaznaczenia wierzchołka jako odwiedzonego, zapisuje się jego numer na pierwsze wolne miejsce w tablicy *ZgadzaSięZ*). W kolejnej fazie tablica jest przeglądana, a dla każdego zapisanego w niej wierzchołka sprawdza się, czy wierzchołki do których prowadzą zeń krawędzie $\xrightarrow{N}$ nie mają ustawionego atrybutu *popierany*. Ten sposób umożliwia szybkie odszukanie ew. krawędzi $\xrightarrow{N}$ pomiędzy elementami zbioru *ZgadzaSię*. Na tym kończy się sprawdzanie wiarygodności jednego świadka. Przed przystąpieniem do podobnej procedury dla następnego, należy ponownie zainicjalizować na **false** atrybut *popierany* oraz opróżnić tablicę *ZgadzaSięZ*. Zauważmy, że atrybut *popierany* wystarczy zainicjować tylko w tych wierzchołkach, których numery znajdują się w tablicy *ZgadzaSięZ*.

Złożoność sprawdzania wiarygodności dla jednego świadka jest rzędu $O(m)$, gdzie m jest liczbą zeznań. Dla każdego z n świadków trzeba ją jednak powtórzyć, co daje złożoność rzędu nm. Gdy każdy świadek wypowiada się o każdym, wówczas $m = n(n-1)$, co daje złożoność rzędu n^3.

Rys. 2 Schemat rozwiązania

```
type
  wsk = ↑sąsiad;
  sąsiad = record
    nr : integer;
    nast : wsk
  end;
  świadek = record
    tak : wsk;
    nie : wsk;
    popierany : boolean;
    niewiarygodny : boolean
  end;
var
  świadkowie : array [1..MaxŚwiadków] of świadek;
  zgadzaSięZ : array [1..MaxŚwiadkow] of integer;
  zIluSięZgadza : integer;

procedure Odwiedź(s : integer);
var
  odwiedzany : wsk;
begin
  świadkowie[s].popierany := true;
  zIluSięZgadza := zIluSięZgadza + 1;
  zgadzaSięZ[zIluSięZgadza] := s;
  odwiedzany := świadkowie[s].tak;
  while (odwiedzany ≠ nil) do begin                pętla (1)
    if not świadkowie[odwiedzany↑.nr].popierany then
      Odwiedź(odwiedzany↑.nr);
    odwiedzany := odwiedzany↑.nast
  end
end;
```

Na rys. 2 przedstawiono schemat algorytmu. Atrybuty wierzchołka oraz zbiory wierzchołków sąsiadujących z danym wierzchołkiem przechowuje się w tablicy *świadkowie*. Zbiór sąsiadów zaimplementowano jako jednokierunkową listę numerów wierzchołków. Dla każdego wierzchołka oddzielnie przechowuje się listę sąsiadów, do których prowadzą krawędzie $\xrightarrow{N}$ (lista *nie*) oraz $\xrightarrow{T}$ (lista *tak*). Informację o tym, czy świadek został uznany za niewiarygodnego przechowuje się dla każdego wierzchołka za pomocą dodatkowego atrybutu *niewiarygodny*.

Powyższy program można nieco poprawić zauważywszy, że jeżeli $a \xrightarrow{T} b$ oraz świadek b został już wcześniej uznany za niewiarygodnego, to a też jest niewiarygodny. Ulepszenie nie zmienia zachowania programu w przypadku pesymistycznym, jednakże w wielu sytuacjach może zapobiec wielokrotnemu przechodzeniu grafu. Wprowadzenie poprawki wymaga zmian w pętlach (1) i (2), opisanych na rys. 3.

Pełen tekst rozwiązania wzorcowego zamieszczono w pliku WIA.PAS na załączonej dyskietce.

UWAGI

W poszukiwaniu metod zmniejszenia rzeczywistego czasu pracy algorytmu można rozważyć tzw. sklejenie

Rys. 2 Schemat rozwiązania — c.d.

```
procedure TestujNaSprzeczność(s : integer);
var
  oskarżany : wsk;
  i : integer;
begin
  i := 1;
  while i < zIluSięZgadza and not (świadkowie[s].niewiarygodny) do
    oskarżany := świadkowie[zgadzaSięZ[i]].nie;
    while (oskarżany ≠ nil) and (notświadkowie[s].niewiarygodny) do begin
      świadkowie[s].niewiarygodny := świadkowie[oskarżany↑.nr].popierany;
      oskarżany := oskarżany↑.nast
    end;
    i := i + 1
  end
end;

begin
  for i := 1 to iluŚwiadkow do świadkowie[i].popierany := false;
  zIluSięZgadza := 0;
  for i := 1 to iluŚwiadkow do begin                    pętla (2)
    Odwiedź(i);
    TestujNaSprzeczność(i);
    AnulujPoparcie
  end;
  for i := 1 to iluŚwiadkow do
  if not świadkowie[i].niewiarygodny then writeln(i)
end;
```

Rys. 3 Usprawnienia rozwiązania

modyfikacja pętli (1):

```
  while (odwiedzany ≠ nil) and not świadkowie[s].niewiarygodny do begin
    if not świadkowie[odwiedzany↑.nr].popierany then Odwiedź(odwiedzany↑.nr);
    świadkowie[s].niewiarygodny := świadkowie[odwiedzany↑.nr].niewiarygodny;
    odwiedzany := odwiedzany↑.nast
  end
```

modyfikacja pętli (2):

```
  for i := 1 to iluŚwiadków do begin
     Odwiedź(i);
     if not świadkowie[i].niewiarygodny then TestujNaSprzeczność(i);
    AnulujPoparcie
  end
```

silnie spójnych składowych grafu. Zauważmy, że jeżeli w grafie krawędzie $\xrightarrow{T}$ tworzą cykl, to obalenie wiarygodności jednego ze świadków w cyklu powoduje obalenie wiarygodności pozostałych (*jeden za wszystkich, wszyscy za jednego*). Podzbiór wierzchołków grafu skierowanego o tej własności, że z każdego jego elementu prowadzi ścieżka do dowolnego innego i odwrotnie, nazywamy **silnie spójną składową**.

Właśnie silnie spójne składowe względem krawędzi $\xrightarrow{T}$ tworzą takie „grupy wspólnej odpowiedzialności".

Optymalizacja polega na poprzedzeniu właściwej fazy algorytmu operacją sklejenia silnie spójnych składowych, tzn. zastąpienia każdej z nich jednym nowym wierzchołkiem. Wszystkie krawędzie, których jednym z końców był jakiś element silnie spójnej składowej, zostają w wyniku operacji podłączone do nowego wierzchołka. Precyzyjny opis algorytmu znajdowania spójnych składowych zainteresowany czytelnik znajdzie w książce [3] (w opisie zadania „Agenci") lub [7].

Powyższa optymalizacja, niestety, nie zmniejsza kosztu działania algorytmu w przypadku pesymistycznym. Jest przy tym o wiele trudniejsza do zaimplementowania, niż optymalizacja polegająca na oznaczaniu niewiarygodnych świadków.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 11 testów (WIA1.IN–WIA11.IN):

- WIA1.IN — tylko jeden świadek, brak zeznań;
- WIA2.IN — test z treści zadania;
- WIA3.IN — trzech świadków, każdy wypowiada się o każdym, brak niewiarygodnych;
- WIA4.IN–WIA6.IN — różne testy poprawnościowe.
- WIA7.IN, WIA8.IN — 400-tu świadków, połowa niewiarygodna.
- WIA9.IN, WIA10,IN — 3000 świadków, połowa niewiarygodna.
- WIA11.IN — 3000 świadków, 8000 zeznań, brak niewiarygodnych.